A DATA-DEPENDENCY BASED INTELLIGENT BACKTRACKING SCHEME FOR PROLOG[1]

Vipin Kumar[2] and Yow-Jian Lin

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-37 September 1987

Abstract

This paper presents a scheme for intelligent backtracking in Prolog programs. Rather than doing the analysis of unification failures, this scheme chooses backtrack points by doing the analysis of data dependency between literals. The other data-dependency based methods previously developed can not be easily incorporated in the Warren's abstract machine, and are not able to perform across-the-clause backtracking intelligently. Our scheme overcomes all these problems. For many problems this scheme is just as effective as intelligent backtracking schemes based upon (more accurate) analysis of unification failure, and yet incurs small space and time overhead. To demonstrate the usefulness of our scheme, we have modified a Warren's abstract machine simulator to incorporate our intelligent backtracking scheme, and have evaluated its performance on a number of problems.

---

[1]To appear in *Journal of Logic Programming*, 1988. This work was supported by Army Research Office grant #DAAG29-84-K-0060 to the Artificial Intelligence Laboratory at the University of Texas at Austin.

[2]Arpanet Address: kumar@sally.UTEXAS.EDU
uucp: ....!seismo!ut-sally!kumar

A Data-Dependency Based Intelligent Backtracking Scheme for Prolog

Vipin Kumar & Yow-Jian Lin
Artificial Intelligence Laboratory
Computer Science Department
University of Texas at Austin

1. Introduction

The backtracking method used in standard Prolog implementations is uninformed; i.e., once a goal fails, backtracking is done to the most recent choice point (that has untried alternatives) even if this choice point has nothing to do with the current failure. This kind of "naive" backtracking can result in a lot of unnecessary search, as the same failure can occur many times before an appropriate choice point is selected. A number of intelligent backtracking schemes have been developed to avoid this kind of redundant backtracking [1], [8], [19], [2], [20]. Most of these approaches perform unification failure analysis to select a backtrack point. This analysis entails substantial overhead, which makes the scope of their practical application rather limited. An intelligent backtracking scheme is useful only if the overhead of the scheme is less than the savings due to reduced backtracking.

In this paper we present a scheme for intelligent backtracking in Prolog programs that is based upon the analysis of data dependency between different literals. A data-dependency-based method was first proposed by Conery & Kibler [7] in the context of AND/OR process model. Lin, Kumar & Leung [18] and Woo & Choe [25] found that Conery's method was incorrect, and presented correct methods for intelligent backtracking. Although these methods are quite suitable in the context of AND/OR process model, they can not be easily incorporated in the Warren's abstract machine (WAM). They also require non-trivial overhead to construct the dependency graph dynamically in the sequential execution of Prolog. Furthermore, these methods construct dependency graphs at the clause level (i.e., a separate dependency graph for each clause); hence they are not able to perform across-the-clause backtracking intelligently. The scheme presented in this paper overcomes all these problems. It (implicitly) maintains a single dependency graph for the whole proof tree, permitting intelligent backtracking across the clause. The overhead for maintaining the implicit graph is minimal, and the scheme easily integrates with WAM.

For many Prolog programs, our scheme is just as effective in eliminating redundant backtracking as other schemes that are based upon (more accurate) analysis of unification failure, and yet it incurs small space and time overhead. To demonstrate the usefulness of our scheme, we

have modified PLM level I simulator [11] (which is a variation of Warren's abstract machine) to incorporate our intelligent backtracking scheme, and have investigated its performance on a number of problems. An earlier version of this paper appears in [16].

Section 2 presents an abstract version of our intelligent backtracking algorithm. Section 3 discusses how this scheme is incorporated in PLM. Section 4 compares our scheme with intelligent backtracking schemes developed by other researchers. Section 5 presents performance results of our scheme on various problems, and analyzes overheads and gains due to the scheme. Section 6 presents a simpler (but less accurate) version of our scheme that has smaller overhead. Section 7 contains concluding remarks and discusses relevance of our approach to parallel execution of logic programs.

2. The Intelligent Backtracking Scheme

2.1 Preliminaries

A literal P is dependent upon those literals that have contributed to the current bindings of variables in the arguments of P. This dependency is captured in a dependency graph in which each node is a literal and there is an arc going from node Q to node P if P is dependent upon Q. Next we discuss how to construct a dependency graph of literals in a proof tree.

In Prolog a variable V is assigned a value (which could be a constant, a structure, a list or some other variable) when some goal[3] P is unified with some clause head. Due to the single assignment property of logic programs, once some value is assigned to V, it is not changed unless failure occurs. (Upon failure a new clause head may be unified with P, which may result in a new value being assigned to V.) Note that variables in the term assigned to V will continue to be assigned values as goals are matched with clause heads during the execution of logic program. Whenever unification of a goal P with some clause head results in assignment of a value to a variable, we attach a tag P to the variable V. (Upon failure if backtracking is done to P, then the assignment of both the tag and the value to V are undone.)

At any time during execution, for any variable V (that has been created) there is a set of goals **generators(V)** that have contributed to the current value of V. This set of goals consists of precisely the tag of V and the tags of other variables that exist in the term assigned to V. If $X_1,...,X_n$ are the variables in the arguments of a goal P, then each literal in {generators(X_i)

---

[3] We use the terms "goal" and "literal" to mean the same thing.

$|1 \le i \le n\}$ is a predecessor of P in the dependency graph. Thus by merely keeping a tag with each variable, we can find predecessors of any literal in the dependency graph.

Fig. 1 shows a Prolog program, and the tags of relevant variables after certain goals have been invoked. For the purpose of illustration, each goal has been given a number. This number is used to tag variables when they become bound during the invocation of the corresponding goal (in reality, the address of the choice point of the goal is used for tagging).

If a goal P fails to unify with any of the clause heads that can possibly match P, then clearly the current values of variables X_i that occur in the arguments of P are not satisfactory. The bindings of variables X_i will not change unless at least one goal in

modifying(P) = {parent of P}[4] ∪ {predecessors of P in the dependency graph}

is unified with a clause head that is different from the current one. Let Q be the most recent goal in modifying(P). Clearly, if backtracking is done to any goal that is more recent than Q, then P would again fail because the values of X_i (the cause of unification failure) would remain unchanged. Hence, in our intelligent backtracking scheme, we directly backtrack to Q and thus skip all the backtracking points that are more recent than Q and less recent than P. Since backtracking to Q alone may not cure the failure of P (because other goals in modifying(P) may be culprits), we pass modifying(P)-{Q} to Q. If later Q fails, then backtracking point is selected from modifying(Q)∪modifying(P)-{Q}. To keep track of the backtracking goals of other relevant goals, we maintain **B-list** with each goal P_i. B-list(P_i) represents those goals that may be able to cure the failure of the goals that have directly or indirectly caused backtracking to P_i. The following is a precise description of our intelligent backtracking scheme.

2.2. The Algorithm

When P_i is invoked for the first time (i.e., when P_i is unified for the first time with the head of a clause), initialize B-list(P_i) to contain the parent of P. Whenever some goal P fails, do the following.

1. Let TEMP = B-list(P)∪{generators(X_i)| $1 \le i \le n$}, where X_i's are the variables that occur in the arguments of P. Select P_m such that P_m is the most recent choice point in TEMP.

---

[4] The parent of P is the goal with whom the head of the clause containing P has been unified.

Figure 1

:- g(A, B)[1].

(1) g(X, Y) :- p(X, Y)[2], q(Y)[3], r(X)[4].

(2) p(U, V) :- e(U)[5], f(V)[6].

(3) e(a).
(4) e(b).

(5) f(g(W)).
(6) f(d).

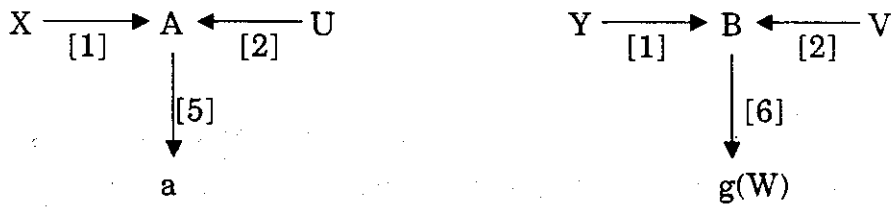
(7) q(g(a)).
(8) q(b).

(9) r(b).

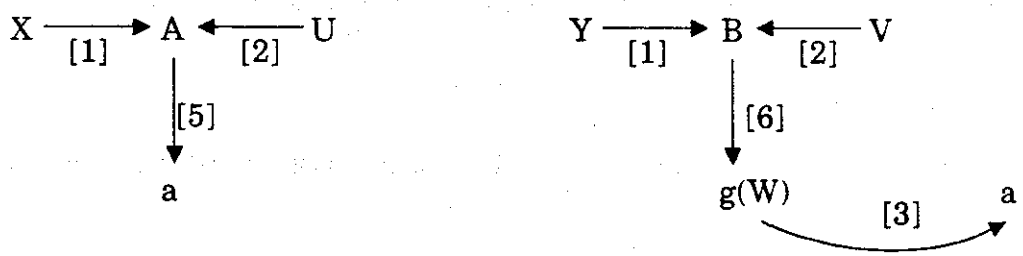
1.a After invoking goal [1]

1.b After invoking goal [2]

1.c After invoking goals [5] and [6]



1.d After invoking goal [3]



2. Add TEMP-{P_m} to B-list(P_m) and backtrack to P_m.

In the program of Figure 1, when r(X) is invoked, it fails. At this time, generators(X) = {[1], [5]} and the parent of r(X) is [1]. Our intelligent backtracking scheme chooses goal [5] (i.e., e(U)) for backtracking, as it is the most recent goal in parent(r(X))∪generator(X). In WAM, the backtracking would be first done to [3] and then to [6] before reaching [5].

3. Implementation details

We have modified PLM level I simulator to implement our intelligent backtracking scheme. PLM [12], [11] is a special-purpose high-performance PROLOG machine based on Warren's abstract machine (WAM) [23]. In the following discussion, familiarity with WAM is assumed.

Whenever we try to solve a goal P (i.e., a literal P), we create a choice point for P. In addition to keeping the information that is normally kept in a choice point, we also keep information about the number of arguments of P, and a pointer to B-list for P. B-list(P) is kept on the heap, and is initialized to contain the parent of P when the choice point for P is created. In the process of unifying P with a clause head, whenever a variable is assigned a value (which could be another variable, constant, list or structure), the variable is tagged with the address of the choice point of P. The tag is kept in a word that follows the content cell. Since, in our modified version of PLM, there is always a unique choice point for a goal that has been solved or that is being solved, we will use P to refer to both the goal and its choice point.

Note that in WAM the choice point for P is created only if there are more than one applicable clauses available for P. Furthermore, in WAM the choice point of P is discarded just before trying the last applicable alternative for P. In our implementation, in both of these cases we have to keep the choice point around, as B-list and arguments of P are needed in case some failure happens to cause backtracking to P.

Whenever failure occurs (in unification), the current (i.e., the most recent) choice point is checked for an alternate clause. If one is available, it is tried. Otherwise intelligent backtracking is invoked as follows. (Note that in the standard WAM, the most recent choice point would definitely have at least one alternate clause available; otherwise it would have been eliminated.) Arguments of the goal P corresponding to the current choice point are inspected to collect generators(X_i) for each variable X_i that occurs in the arguments of P. These are put into the B-list of P in such a way that B-list remains ordered (i.e., the most recent choice point comes first) and contains no duplications. Head P_m of the modified B-list(P) is chosen for backtracking, and the

remaining part of B-list(P) is merged into B-list(P_m).

Implementing CUT

In PLM whenever CUT is encountered in the body of a clause, all the choice points from the top of the stack to P (where P is the goal that was unified with the head of the clause containing CUT) are eliminated. In our implementation, they can not be simply eliminated, as they may have been used as tags for variables that may occur in the arguments of goals appearing after the CUT symbol. CUT can be handled in our scheme if one of the following is done after encountering CUT.

1. For every variable that has been assigned a value after the invocation of P (where P is the parent goal of CUT), change its tag to P. This can be done by maintaining another TRAIL like stack to keep track of all the variables that have been assigned values. (TRAIL in WAM keeps track of only some variables.) Now eliminate all the choice points between CUT and P.
2. Mark all the choice points between CUT and P with a special flag. In future, if backtracking is done to any such choice point, then eliminate all choice points until P.

We have chosen to implement the second alternative, as it was easier to implement, and it also requires less overhead in terms of CPU time. Note that the generator/consumer approach for finding culprits for a failure works fine as long as CUT is used only as a device for saving search space. If CUT is used to perform non-monotonic reasoning, then it is possible that some solutions can be missed by this scheme. Consider the following example:

goal: :- p(X,T), q(X).

Program:

```
1)  p(Y,a).
2)  p(c,b).
3)  q(Z)   :-r(Z,U), !, s(U).
4)  r(b,a)
5)  r(c,b).
6)  s(b).
```

After p(X,T) is successfully unified with the head of the clause 1, T is bound to a, Y is bound to X, and X remains unbound. When the execution continues, q(X) fails. At this point generator(X)

= nil, and B-list(q(X)) = {top goal}. Hence backtracking is done to the top goal, and the program halts without finding any solution. If the clauses used to solve q(X) are pure Horn Clauses, then the result would be correct because if the execution of q(X) cannot succeed for unrestricted value of X, it should not succeed for any specific value of X. However, since clause 3 has CUT, it is no longer a pure Horn Clause. In the above example, q(X) succeeds if X = c, as clause 4 would be avoided in solving r(Z,U) and a solution can be generated using clauses 2, 3, 5, and 6.

The problem happened because, for a goal p, generator(X) contains only those goals that have made any non-variable binding to the variables X in the arguments of p. The goals that "couple" variables X in the arguments of p with other variables (and hence have the potential of changing the bindings of X via other clauses) are not in generator(X). If the proof tree underneath p has nonlogical operators such as CUT or NOT, then p should become dependent upon all those goals that are "coupled" with the variables X in the argument of p; i.e. goals that have changed the bindings of these variables as well as those goals that could have changed them. Let's designate this new set as **coupled(X)**; i.e., the set of goals that are coupled to variable X. Clearly, if a goal has a non-monotonic operator underneath, then for each variable X in the argument of p, we need to compute coupled(X) instead of generator(X) to construct modifying(p). Note that the data structure that will permit the computation of coupled(X) is a bit more complicated. Our current implementation does not create such data structure. Therefore, it can only execute those programs that use CUT primarily for the purpose of saving search space.

4. Comparison with Related Research

4.1 Data-dependency Based Methods

The data-dependency based methods of Lin, Kumar & Leung [18] and Woo & Choe [25] use dependency graphs at the clause level. These graphs are already available (from the forward execution) if AND parallelism is being executed in the framework of AND/OR process model. In principle, the same technique can be used to perform intelligent backtracking in the sequential execution of Prolog. But constructing dependency graphs at the clause level (in the sequential execution) can be very expensive, and it is awkward to incorporate the clause-level backtracking scheme in WAM. The scheme presented in this paper was discovered when we were trying to find an efficient sequential implementation of our previous scheme. The main feature of the new scheme is that it uses one dependency graph for the whole proof tree. As we saw in Section 2, this graph is implicitly available if we keep a tag with each variable. Since this method performs backtracking at the proof-tree level, it integrates well with WAM and it is more powerful than clause-level data dependency based backtracking schemes. This is illustrated in the following

5. Performance Results

We have tested our scheme on a variety of problems. To compare the performance of our scheme with other schemes [3], [2], we have run it on the same programs that were used to evaluate these schemes.

We have modified the PLM level I simulator to provide us the number of machine cycles required by each PLM instruction. The computation of machine cycles is based upon the information provided in [11], [12]. For each program, we have collected the following three figures: (i) the CPU time taken by the simulator; (ii) the number of PLM instructions executed; (iii) the number of machine cycles executed. Of the three, only the number of machine cycles can give us an accurate picture, as the CPU time consumed is greatly determined by the simulator overhead (in addition to the overhead due to intelligent backtracking), and different machine instructions can consume varying amount of CPU time. Hence, we can only provide a rough comparison of our scheme with the schemes in [3], [2], as in [2] only the CPU time taken by their interpreter was reported, and in [3], only the number of PLM instructions executed was reported.

These results and the results presented in [3], [2] are summarized in Table I. Columns II and III show CPU time taken, number of machine instructions, and number of machine cycles for naive and our intelligent backtracking schemes. Column IV shows the % difference between naive and our approach in terms of CPU time, machine instructions and machine cycles. Column V shows the difference between naive backtracking and Chang & Despain's semi-intelligent backtracking scheme in terms of number of instructions as reported in [3]. Column VI shows the difference between naive backtracking and Bruynooghe & Pereira's scheme in terms of CPU time as reported in [2].

As seen from Table I, the performance of our scheme in terms of CPU time is consistently better than in terms of the number of machine cycles. This happens because the overhead of running the PLM simulator is quite large, and it tends to overshadow the overhead due to intelligent backtracking. In contrast, PLM is a high performance Prolog architecture; in terms of the number of machine cycles executed, it is extremely efficient (see [11]). Hence even small overhead due to intelligent backtracking immediately shows up. We can expect the same phenomenon to occur in the experiments of Bruynooghe & Pereira; i.e., since their naive interpreter most likely has a large overhead (compared with the PLM compiler with naive backtracking), the overhead caused by their intelligent backtracking scheme appears smaller than it really is.

Our scheme always performs better than Bruynooghe & Pereira's scheme except for the simple program for solving queens problem. For this program, generator/consumer analysis is not precise enough to eliminate any backtracking. It is not possible to provide a direct

Table I

	Naive PLM			Our Intelligent Backtracking			Change			Chang, et.al	B & P
	CPU	#M	#C	CPU	#M	#C	CPU	#M	#C	#M	CPU
query	6.46	1310	14174	2.88	549	7249	-55.42%	-58.09%	-48.86%	-16%	-20%
binary tree	1.22	1297	11047	1.22	1297	12041	0.00%	0.00%	9.00%	-	44%
6queens(clever)	37.20	29126	240044	40.38	28466	457511	8.55%	-2.27%	90.59%	-	99%
6queens(simple)	65.80	69782	465081	76.74	69486	1222326	16.63%	-0.42%	162.82%	-	-36%
mapcolor(clever)	2.58	897	10467	2.48	859	11283	-3.88%	-4.24%	7.80%	0.7%	63%
mapcolor(bad)	8113.24	2484865	32656404	5.70	2036	26016	-99.93%	-99.92%	-99.92%	-99.9%	-99.7%
circuit design	82.20	51645	503877	24.32	14421	199924	-70.41%	-72.08%	-60.32%	-	-

CPU refers to the CPU time taken (in seconds);

#M refers to the number of PLM instructions executed;

#C refers to the number of machine cycles.

The program for Circuit Design is shown in Appendix A.

comparison of our approach with Chang & Despain's static backtracking scheme, as some instructions in our modified PLM take more machine cycles (because of intelligent backtracking overhead) than similar instructions in PLM. (Our scheme always does better in terms of the number of PLM instructions executed.) But the overhead of Chang and Despain's approach is very small, as the data-dependency analysis is done at the compile time. As discussed in Section 4.2, the major problem with their approach is that it is very conservative. The performance results for the query program clearly show that our scheme is much more precise than Chang & Despain's scheme and has much smaller overhead than Bruynooghe & Pereira's scheme (see Table I).

Overheads and Gains due to our Intelligent Backtracking Scheme

The overheads due to our scheme can be categorized into five parts.

1. The overhead due to variable tagging.
2. The overhead for traversing the variable bindings of the arguments of failed goals p to construct modifying(p).
3. The overhead for manipulating (inserting, merging, etc.) B-lists.
4. The overhead for creating and maintaining a choicepoint frame even if the goal has only one alternative. (See Section 3).
5. Miscellaneous overhead. Part of this is the overhead due to extra checking done at each failure. These checks are done to recognize whether the failure is shallow or deep, and whether backtracking is being done into the scope of cut. The overhead due to CUT is also included in this category. (See Section 3).

Of all these overheads the first two are specific to our scheme, and the last three will normally be incurred by any approach that does some analysis to find culprits of the failure. The size of the space searched by each approach (naive and intelligent) is computed by counting the number of unifications between goals and clause heads. Since each unification can take different amount of time, this only provides a rough figure. This information is summarized in Table II a-g.

The overhead due to tagging is consistently very small (less than 1.3%) in all the programs we tried, as the machine cycles for tagging are usually overlapped with cycles for other activities. This shows that the extra information necessary to find generator/consumer relationship in our approach can be kept at a very small cost.

In a fully deterministic program (such as the program for binary tree), the overheads for constructing modifying(p) and for manipulating B-lists are nil, as there is no deep failure in such

Table II

(a) Database Problem

	Naive	Intelligent	Change
Machine Cycle	14174	7249	-48.86%
Search Space	356	128	-64.04%
Overheads (in machine cycles)			
Tagging		55	0.76%
Creating Special CP		174	2.40%
Manipulating B-list		554	7.64%
Traversing Arguments		226	3.12%
Misc. Cost		780	10.76%

(b) Binary Tree Problem

	Naive	Intelligent	Change
Machine Cycle	11047	12041	9.00%
Search Space	217	217	0.00%
Overheads (in machine cycles)			
Tagging		37	0.31%
Creating Special CP		471	3.91%
Manipulating B-list		0	0.00%
Traversing Arguments		0	0.00%
Misc. Cost		486	4.04%

(c) 6-queen Problem (clever program)

	Naive	Intelligent	Change
Machine Cycle	240044	457511	90.59%
Search Space	5174	5009	-3.19%
Overheads (in machine cycles)			
Tagging		1150	0.25%
Creating Special CP		3957	0.86%
Manipulating B-list		142567	31.16%
Traversing Arguments		49272	10.77%
Misc. Cost		27320	5.97%

(d) 6-queen Problem (simple program)

	Naive	Intelligent	Change
Machine Cycle	465081	1222326	162.82%
Search Space	9208	9134	-0.80%
Overheads (in machine cycles)			
Tagging		5489	0.45%
Creating Special CP		43379	3.55%
Manipulating B-list		477940	38.75%
Traversing Arguments		173123	14.04%
Misc. Cost		52188	4.27%

(e) Map Coloring Problem (clever)

	Naive	Intelligent	Change
Machine Cycle	10467	11283	7.80%
Search Space	257	242	-5.84%
Overheads (in machine cycles)			
Tagging		28	0.25%
Creating Special CP		14	0.12%
Manipulating B-list		84	0.74%
Traversing Arguments		54	0.48%
Misc. Cost		1230	10.90%

(f) Map Coloring Problem (bad)

	Naive	Intelligent	Change
Machine Cycle	32656404	26016	-99.92%
Search Space	804059	505	-99.94%
Overheads (in machine cycles)			
Tagging		66	0.25%
Creating Special CP		280	1.08%
Manipulating B-list		483	1.86%
Traversing Arguments		164	0.63%
Misc. Cost		2396	9.21%

(g) Circuit Design Problem

	Naive	Intelligent	Change
Machine Cycle	503877	199924	-60.32%
Search Space	7342	2009	-72.64%
Overheads (in machine cycles)			
Tagging		2574	1.29%
Creating Special CP		1560	0.78%
Manipulating B-list		32291	16.15%
Traversing Arguments		15257	7.63%
Misc. Cost		11157	5.58%

programs. In nondeterministic programs (such as map coloring, n-queens, database query, circuit design), these overheads vary depending upon how frequently deep failures occur and how large the structures bound to the variables in the arguments of the failed literals are. Clearly, our scheme incurs substantial overhead in these two categories if the structures bound to variables are large. Furthermore, in such cases, the analysis of failure becomes less precise. This explains the poor performance (both in terms of reduction in search space and in terms of overhead) of our scheme on the 6-queen programs. For the other nondeterministic programs, our scheme is able to reduce the search space and (with the exception of the clever program for map coloring) the number of machine cycles required. For the clever program for map coloring, the reduction in search space is not big enough to overcome the overheads.

The overhead for creating extra choice point is very small for all the programs except for the binary tree program and the simple program for the 6-queen problem. For these two programs, the intelligent backtracking scheme creates (special) choice points for many goals that have only one alternative. Miscellaneous overhead is in the range of 4-11% depending upon how frequently failures occur and how many times CUT is encountered.

The machine cycle count provides an accurate picture of the overhead of the current implementation of our backtracking scheme. It should be possible to tune this implementation to reduce the overhead. Appropriate modification to the PLM machine architecture to support certain activities needed by the intelligent backtracking scheme (such as a separate area for maintaining B-list, and specialized instructions to construct and manipulate the new choice points) would further reduce the overhead. Hence, the overhead of our intelligent backtracking scheme in terms of machine cycles should only be looked as an upper bound.

Is Intelligent Backtracking Really Needed?

In many programs, if the ordering of literals is changed, then naive backtracking does little redundant search and becomes just as good as intelligent backtracking. This effect is clearly seen in the map coloring programs used in our experiments. The simple and clever programs for map coloring differ only in the ordering of literals, and for the clever program, naive and intelligent backtracking have comparable performance. Even in the query program and the circuit design program, a change in the ordering of literals makes intelligent backtracking unnecessary. This makes one wonder whether intelligent backtracking is really needed.

To see that there are programs for which intelligent backtracking helps irrespective of ordering, consider the program in Figure 2. We tried all possible (two) orderings of the main clause. (The ordering in the other clauses does not matter much.) For each ordering, intelligent backtracking searches a smaller space than the naive backtracking. Note that in the map-coloring

Figure 2

This program checks whether a move generated by predicate *move* is a good move. *move(X,Y,Z)* is good if both X and Z movement w.r.t. Y are legal, and the number of steps in checking X w.r.t. Y is greater than that in checking Z w.r.t. Y.
A legal movement *legal(X,Y,N)* is defined as follows:

1. The movement in X is at least three steps more than that in Y; and
2. Either the movement in X is at least five steps more than that in Y, or the movement in X is twice of that in Y after moving in both X and Y simultaneously no more than ten times (one step at a time).

legal(X,Y,N) returns N as the number of steps needed to become true.

```
goal :- move(X,Y,Z), legal(X,Y,A), legal(Z,Y,B), A > B, write([X,Y,Z,A,B]).

legal(X,Y,N)     :- legal1(X,Y), legal2(X,Y,N).

legal1(X,Y)      :- X >= (Y+3).

legal2(X,Y,1)    :- X > (Y+5).
legal2(X,Y,N)    :- check(X,Y,1,K), N is K+1.

check(X,Y,N,N) :- X is Y+Y, !.
check(X,Y,N,K) :- N < 10, X1 is X+1, Y1 is Y+1, N1 is N+1, check(X1,Y1,N1,K).

move(3,1,7).     move(3,3,9).     move(15,3,5).    move(11,5,4).    move(1,3,9).
move(20,7,8).    move(30,9,6).    move(12,6,2).    move(5,4,10).    move(8,2,3).
move(2,8,18).    move(12,4,4).    move(5,7,15).    move(8,4,10).    move(3,9,19).
```

Ordering 1:

```
goal :-  move(A,B,C), legal(A,B,X), legal(C,B,Y), X > Y, write([A,B,C,X,Y]).
```

Ordering 2:

```
goal :-  move(A,B,C), legal(C,B,Y), legal(A,B,X), X > Y, write([A,B,C,X,Y]).
```

Ordering	Matches Tried			Machine Cycles		
	Naive	Intelligent	Change	Naive	Intelligent	Change
1	382	104	-72.77%	12222	5606	-54.13%
2	253	99	-60.87%	8199	5421	-33.88%

program, variation in the number of machine cycles due to different ordering is much less for intelligent backtracking (1:2.4) than for naive backtracking (1:3120) (see Table I). Since in more complex constraint satisfaction programs, a good ordering may not be obvious, a low overhead intelligent backtracking (such as ours) can be very valuable. In generate & test type programs (such as the program in Figure 2), reordering may not help at all unless the program is completely rewritten.

6. A Simpler Backtracking Scheme

It is possible to simplify our intelligent backtracking scheme and reduce the overheads even further (at the risk of making the backtracking scheme less accurate). Let's classify failures into two classes: Type I and Type II. A failure is Type I if the literal fails to return a solution for the first time it is called. A failure is Type II if the literal had succeeded earlier but fails to find another solution. These definitions are very similar to the definitions of Type I & II backtrackings given in [3].

A simple version of our scheme is obtained if we only perform Type I backtrackings intelligently (and Type II backtrackings naively). In this case, we no longer need to keep track of B-lists - once a backtracking literal is chosen, the other candidates can be discarded. Another significant advantage is that now we can choose to perform Type I intelligent backtracking only for selected goals (that are suspected to benefit from it) and avoid paying overheads for creating a special choice point for the remaining (naive) literals. We found that for many problems (map coloring, circuit design), intelligent Type I backtracking is nearly as effective as Type I and II combined.[5] For the circuit design problem, selective Type I backtracking performs better than Type I & II, even though Type I backtracking results in more search (see Table III(a)). But in general, intelligent Type I backtracking alone can perform much worse than Type I & II combined (Table III(b)).

---

[5] For some constraint satisfaction problems, Dechter found that simple Type I intelligent backtracking (she calls it BACKJUMP) is nearly as effective as many other complicated schemes) [9].

Table III

	Type I & Type II	**Type I only** (for all goals)	**Type I only** (for selected goals)
Machine Cycle	199924	206378	187600
Search Space	2009	2406	2406
Overheads (in machine cycles)			
Tagging	2574	3145	2994
Creating Special CP	1560	5189	3422
Manipulating B-list	32291	12255	3131
Traversing Arguments	15257	8122	2042
Misc. Cost	11157	12373	12166

(a) Performance results of Type I vs Type I & II backtracking scheme on Circuit Design program.

	Type I & Type II	**Type I only** (for all goals)	**Type I only** (for selected goals)
Machine Cycle	7249	12686	12259
Search Space	128	231	231
Overheads (in machine cycles)			
Tagging	55	103	103
Creating Special CP	174	697	573
Manipulating B-list	554	327	160
Traversing Arguments	226	238	80
Misc. Cost	780	1402	1424

(b) Performance results of Type I vs Type I & II backtracking scheme on query program.

7. Concluding Remarks

We have presented a scheme for intelligent backtracking in logic programs. This scheme uses data-dependency relationships between literals to perform backtracking intelligently. Our implementation of this scheme has shown that intelligent backtracking can be incorporated in Warren's abstract machine without causing excessive overhead. Some of the problems that appear suitable for our intelligent backtracking scheme are Constraint satisfaction problems and the problems solved by the generate-and-test paradigm.

Since the overhead of our scheme is still non-trivial (at least for some programs), it should only be used if it appears that the program may benefit from intelligent backtracking. Since there should also be problems for which the added power of unification based backtracking schemes is able to compensate for their overhead, it would be desirable to have many compilers incorporating different intelligent backtracking schemes available at the user's discretion.

Our intelligent backtracking scheme can be used if the AND-parallelism is exploited in the generator/consumer framework [7], [4], [17], [10], [14]. In the generator/consumer framework, a goal is allowed to execute only if it shares no unbound variable with any other executing goal. This guarantees the consistency of the dependency graph even if it is concurrently manipulated by many processors. The scheme can also be used if OR-parallelism is exploited in the context the AND/OR process model [7], [21]. If full OR-parallelism is exploited (as in [15], [5]) then all possible paths are explored, and none of the intelligent backtracking schemes are needed.

Acknowledgement:

The authors would like to thank Al Despain and Barry Fagin of University of California, Berkeley for providing the code of PLM level I simulator and the program in Appendix A. The authors would also like to thank Manuel Hermenegildo, Marc Stickel, Jonas Barkland, Madhur Kohli and anonymous reviewers for many useful comments on an earlier draft of this paper.

REFERENCES

[1] M. Bruynooghe, Intelligent Backtracking for an Interpreter of Horn Clause Logic Programs, *Mathematical Logic in Computer Science*, Salgotarjan, Hungary, Colloquia Mathematica Societatis Janos Bolyai, 1978.

[2] M. Bruynooghe and L. M. Pereira, Deduction Revision by Intelligent Backtracking, pp. 194-215 in *Implementations of Prolog*, ed. J. A. Campbell, Ellis Horwood Limited, 1984.

[3] J.-H. Chang and A.M. Despain, Semi-Intelligent Backtracking of Prolog Based on a Static Data Dependency Analysis, *Proceedings of IEEE Symposium on Logic Programming*, pp. 10-21, August, 1985.

[4] J.-H. Chang, A.M. Despain, and D. Degroot, AND-Parallelism of Logic Programs Based on Static Data Dependency Analysis, *Proceedings of the 30th IEEE Computer Society International Conference*, pp. 218-226, February, 1985.

[5] A. Ciepielewski and S. Haridi, A Formal Model for OR-Parallel Execution of Logic Programs, in *Proceedings of IFIP 83*, ed. Mason, North Holland P.C., 1983.

[6] C. Codognet, P. Codognet, and G. File, Depth-first Intelligent Backtracking, Technical Report, Mathematiques et Informatique, Universite de Bordeaux, 1986.

[7] J.S. Conery and D.F. Kibler, AND Parallelism and Nondeterminism in Logic Programs, *New Generation Computing 3(1985)*, pp. 43-70, OHMSHA,LTD. and Springer-Verlag, 1985.

[8] P. Cox and T. Pietrzykowski, Deduction Plans: a basis for intelligent backtracking, *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-3, 1*, 1981.

[9] R. Dechter, Learning While Searching in Constraint Satisfaction Problems, *Proc. of Fifth National Conf. on Artificial Intelligence (AAAI-86)*, pp. 178-185, August 1986.

[10] D. Degroot, Restricted AND-Parallelism, *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, pp. 471-478, 1984.

[11] T. Dobry, A. Despain, and Y. Patt, Performance Studies of a Prolog Machine Architecture, *Proceedings of IEEE Symposium on Logic Programming*, pp. 180-190, August, 1985.

[12] T. Dobry, Y. Patt, and A. Despain, Design Decisions Influencing the Microarchitecture for a Prolog machine, *MICRO 17 Proceedings*, October 1984.

[13] J. Doyle, A Truth Maintenance System, *Artificial Intelligence 12*, pp. 231-272, 1979.

[14] M. Hermenegildo, An Abstract Machine for Restricted AND-parallel Execution of Logic programs, *Third Int'l Conf on Logic Programming*, July 1986.

[15] L. V. Kale, Parallel Architectures for Problem Solving, Ph.D. Dissertation,, Computer Science Department, SUNY at Stony Brook, Stony Brook, NY, December, 1985.

[16] V. Kumar and Y. J. Lin, An Intelligent Backtracking Scheme for Prolog, *1987 Symposium on Logic Programming, September 1987*, also appeared as Tech. Report #86-41 (December 1986), AI Lab, University of Texas at Austin, Austin, Texas 78712.,

[17] Y.J. Lin and V. Kumar, A Parallel Execution Scheme for Exploiting AND-parallelism of Logic Programs, *the 1986 International Conference on Parallel Processing*, St. Charles, Illinois, August, 1986.

[18] Y.J. Lin, V. Kumar, and C. Leung, An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs, *the Third International Conference on Logic Programming*, London, England, pp. 55-68, July, 1986.

[19] S. Matwin and T. Pietrzykowski, Intelligent Backtracking in Plan based deduction, *IEEE Trans. on PAMI 7*, 6, November 1985.

[20] L. M. Pereira and A. Porto, Selective Backtracking, pp. 107-114 in *Logic Programming*, ed. K. L. Clark and S.-A. Tarnlund, Academic Press, 1982.

[21] N. Tamura and Y. Kaneda, Implementing Parallel Prolog on a Multiprocessor Machine, *Proceedings of IEEE Symposium on Logic Programming*, Atlantic City, pp. 42-48, February, 1984.

[22] J. Toh and K. Ramamohanrao, Failure Directed Backtracking, Tech. Report 86/9, Computer Science Dept., University of Melbourne, Australia, 1986.

[23] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI International, October, 1983.

[24] David Wolfram, Intractable Unifiability Problems and Backtracking, *Third Int'l Conf. on Logic Programming*, pp. 107-121, July 1986.

[25] N. Woo and K. Choe, Selecting the Backtrack Literal in the AND/OR Process Model, *1986 Symposium on Logic Programming*, Salt Lake City, pp. 200-209, 1986.

Appendix A: The program for Circuit Design

The program designs a combinatorial circuit with three inputs (S, A, B) and one output (out). Input list is the truth table for the function to be synthesized. For example, input list [0,0,1,1,0,1,0,1] means SAB = 000, out = 0, SAB = 001, out = 0, etc..

```
main :- run(0, [0,0,1,1,0,1,0,1], L), write([ circuit,=,L]), nl.

run(Depth, Table, Circuit) :- t(Depth, Circuit, Table).
run(Depth, Table, Circuit) :- D is Depth + 1, run(D, Table, Circuit).

t(_, 0 , [0,1,0,1,0,1,0,1]).
t(_, 1 , [0,0,1,1,0,0,1,1]).
t(_, 2 , [0,0,0,0,1,1,1,1]).
t(_, i0 , [1,0,1,0,1,0,1,0]).
t(_, i1 , [1,1,0,0,1,1,0,0]).
t(_, i2 , [1,1,1,1,0,0,0,0]).
t(Depth, [i,Z], Table)     :- Depth > 0, D is Depth -1, sint(Table, Itable), t(D, Z, Itable).
t(Depth, [n,Y,Z], Table) :- Depth > 0, D is Depth -1, ngate(Table, A, B), t(D,Y,A), t(D,Z,B).

sint([],[]).
sint([X|T1],[_|T2]) :- var(X), sint(T1, T2),!.
sint([0|T1],[1|T2]) :- sint(T1, T2).
sint([1|T1],[0|T2]) :- sint(T1, T2).

ngate([], [], []).
ngate([X|T0], [_|T1], [_|T2])      :- var(X), !, ngate(T0, T1, T2).
ngate([X|T0], [1|T1], [1|T2])      :- X==0, ngate(T0, T1, T2).
ngate([X|T0], [_|T1], [0|T2])      :- X==1, tgate(T0, T1, T2).

tgate([], [], []).
tgate([X|T0], [_|T1], [_|T2])      :- var(X), !, tgate(T0, T1, T2).
tgate([X|T0], [1|T1], [1|T2])      :- X==0, tgate(T0, T1, T2).
tgate([X|T0], [_|T1], [0|T2])      :- X==1, tgate(T0, T1, T2).
tgate([X|T0], [0|T1], [_|T2])      :- X==1, tgate(T0, T1, T2).
```